Algorithms and Complexity

4. Dynamic Programming

Marc Hellmuth

University of Stockholm

Dynamic Programming (DP)

Dynamic Programming is . . .

- . . . a general, powerful algorithm design technique for solving optimization problems.
- . . . a type of “very smart” exhaustive search that can be applied when the problem can be “subdivided” into overlapping subproblems.
- . . . solves problems by combining the solutions to subproblems
- . . . computes the value of an optimal solution first. Optionally, the optimal solution can be constructed from computed information (backtracking).

Example: Fibonacci Numbers...

Sequence: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

... are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

Example: Fibonacci Numbers. . .

Sequence: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

. . . are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way:

F(positive integer n)

if $n \leq 2$ **then** $f = 1$
else
    $f = F(n-1) + F(n-2)$
return f

Example: Fibonacci Numbers...

Sequence: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

...are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way:

F(positive integer n)

if $n \leq 2$ **then** $f = 1$
else
    $f = F(n-1) + F(n-2)$
return f

recursive way with memo:

F(positive integer n)

if memo[n] $\neq$NIL **then**
    return memo[n]
if $n \leq 2$ **then** $f = 1$
else
    $f = F(n-1) + F(n-2)$
memo[n] = f
return f

Example: Fibonacci Numbers. . .

Sequence: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

. . . are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way:

F(positive integer n)

if $n \leq 2$ **then** $f = 1$
else
    $f = F(n-1) + F(n-2)$
return f

recursive way with memo:

F(positive integer n)

if $\mathtt{memo}[n] \neq$NIL **then**
    return $\mathtt{memo}[n]$
if $n \leq 2$ **then** $f = 1$
else
    $f = F(n-1) + F(n-2)$
$\mathtt{memo}[n]$ = f
return f

Which algorithm is more efficient and why? WHITEBOARD

Dynamic Programming (DP)

In general, the design of DP consists of the following steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, (typically in a bottom-up fashion).
4. Construct an optimal solution from computed information (backtracking)

Floyd-Warshall Algorithm (WHITEBOARD)

Aim: Find shortest $u - v$ path for all $u, v \in V$ for given di-graph $G = (V, E)$ with conservative weighting $w : E \to \mathbb{R}$.

- W.l.o.g. $V = \{1, 2, \ldots, n\}$
- $V^k := \{1, 2, \ldots, k\}$, $k \leq n$
- Inner vertices of path $< v_0, v_1, \ldots v_{k-1}, v_k >$ are $v_1, \ldots v_{k-1}$
- $d_{ij}^{(k)}$ is the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set V^k, where $d_{ij}^{(0)} = w(i,j)$ if edge (i,j) exists
- matrix W with

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{else if} (i,j) \in E \\ \infty & \text{else, i.e., } i \neq j, (i,j) \in E \end{cases}$$

-

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & \text{if } k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{else, i.e., } k \geq 1 \end{cases}$$

Because for any path, all intermediate vertices are in the set $V^n = V$, the matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer: $d_{ij}^{(n)} = \delta(i,j)$ for all $i, j \in V$.

Floyd-Warshall Algorithm

FLOYD-WARSHALL(matrix W, n)

$D^{(0)} = W$
for $k = 1, \ldots, n$ **do**
  let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
  for $i = 1, \ldots, n$ **do**
    for $j = 1, \ldots, n$ **do**
      $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
return $D^{(n)}$

Theorem 4.1

Let $G = (V, E)$ be a digraph with conservative weighting w. Then, FLOYD-WARSHALL *correctly computes the distances between all vertices of G in $O(|V|^3)$-time.*

Backtracking to find shortest paths: WHITEBOARD

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$ = sequence of letters
- $Z = z_1 z_2 \dots z_k$ is subsequence of $X = x_1 x_2 \dots x_m$, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$
  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS) (WHITEBOARD)

LCS(strings X, Y)

m= X.length, n= Y.length
Let $b[1\dots m,1\dots n]$ be new array
Let $c[0\dots m;0\dots n]$ be new array
for $i=1\dots m$ **do** $c[i,0]=0$
for $j=0\dots n$ **do** $c[0,j]=0$
for $i=1\dots m$ **do**
    for $j=1\dots n$ **do**
        if $x_i=y_j$ **then**
            $c[i,j]=c[i-1,j-1]+1$
            $b[i,j]=\text{``}\nwarrow\text{''}$
        else if $c[i-1,j]\geq c[i,j-1]$ **then**
            $c[i,j]=c[i-1,j]$
            $b[i,j]=\text{``}\uparrow\text{''}$
        else
            $c[i,j]=c[i,j-1]$
            $b[i,j]=\text{``}\leftarrow\text{''}$
return c and b

PRINT_LCS(b, X, i,j)
// Initial call PRINT_LCS(b, X, m,n)

if $i=0$ or $j=0$ **then return**
if $b[i,j]=\text{``}\nwarrow\text{''}$ **then**
    PRINT_LCS(b, X, $i-1$,$j-1$)
    print x_i
else if $b[i,j]=\text{``}\uparrow\text{''}$ **then**
    PRINT_LCS(b, X, $i-1$,j)
else
    PRINT_LCS(b, X, i,$j-1$)

Theorem 4.3

LCS( ) *and* PRINT_LCS( ) *correctly returns length and LCS of two strings* $X=x_1\dots x_n$ *and* $Y=y_1\dots y_m$ *in* $O(mn)$ *time.*